# 基于深度学习的目标检测与识别技术报告

HRT19D-detection

寒假培训第一次实验

制作：王昱豪，赫子龙，李昊天

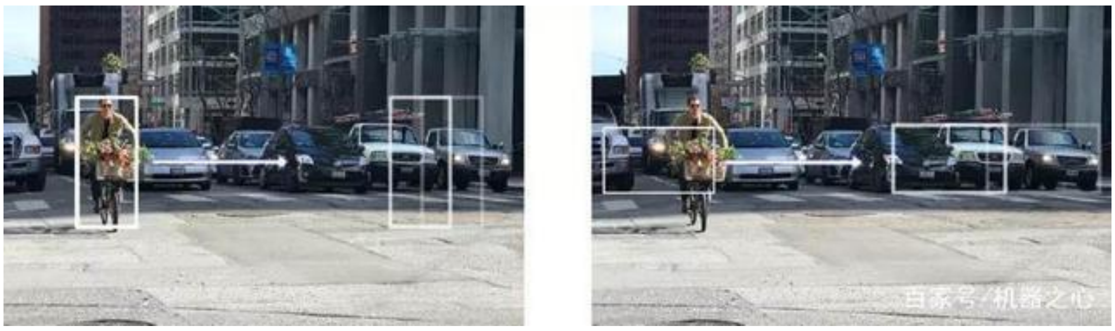
2019.01.15

# 前言

本次实验的课题是目标检测与识别技术，目前计算机视觉方面主要有基于候选区域的目标检测器以及单次检测器，而单次检测器在小目标的检测上通常表现很差。虽然在 18 赛季车队的目标检测技术使用的是单次检测器的 YOLO v3，其原因是 YOLO v3 的响应速度更快，可以直接实现端对端。但鉴于中国赛的部分赛制规则（可重复经过同一赛道并在第一圈不计时），我们认为可以使用基于候选区域的目标检测器以提高精度。在现领域的基于候选区域的目标检测器中比较好的是 R-CNN 系列。所以，我们选择 R-CNN 系列作为研究对象。

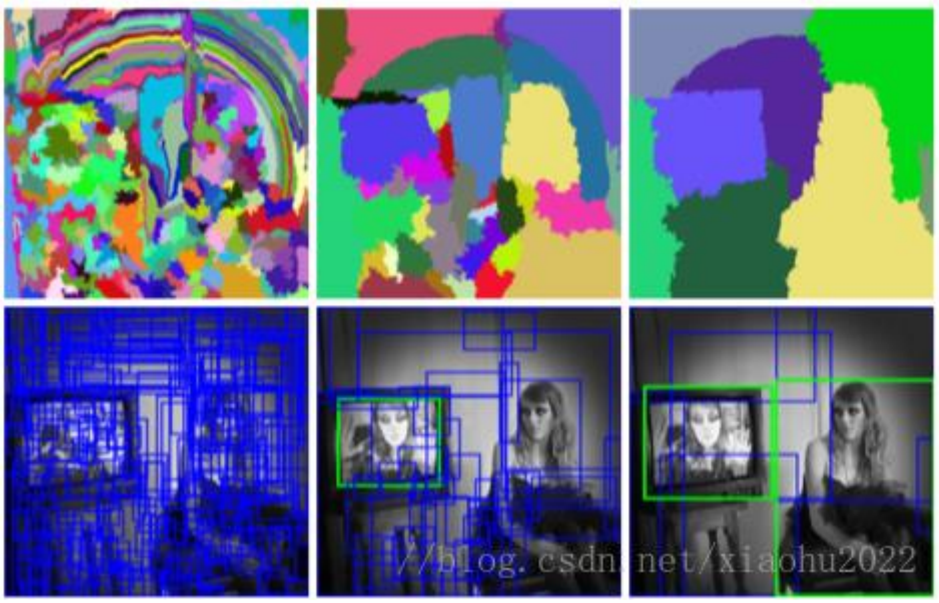
## Part 1 基本原理

要想了解 R-CNN，首先要了解滑动窗口检测器。这是一种暴力的目标检测方法，使用不同大小、不同宽高比的窗口，从左到右、从上到下滑动，剪切图像块交给分类器处理。显然，这其中的滑动步幅与图像窗口数量形成了对立：大步幅必然造成检测精度的大幅下降，而小步幅所带来的图像大量增多也必然造成计算资源的大量需求与计算时间的大幅延长。



**滑动窗口检测器工作原理**

至此，候选区域方法应运而生。候选区域方法在滑动窗口的基础上，先通过简单的区域划分算法将图片划分成很多小区域，再通过层级分组方法根据一定的相似度来合并它们，最终剩下的就是候选区域。这种方式大大减少了需处理的图像数量，减少了计算资源使用和计算时间。



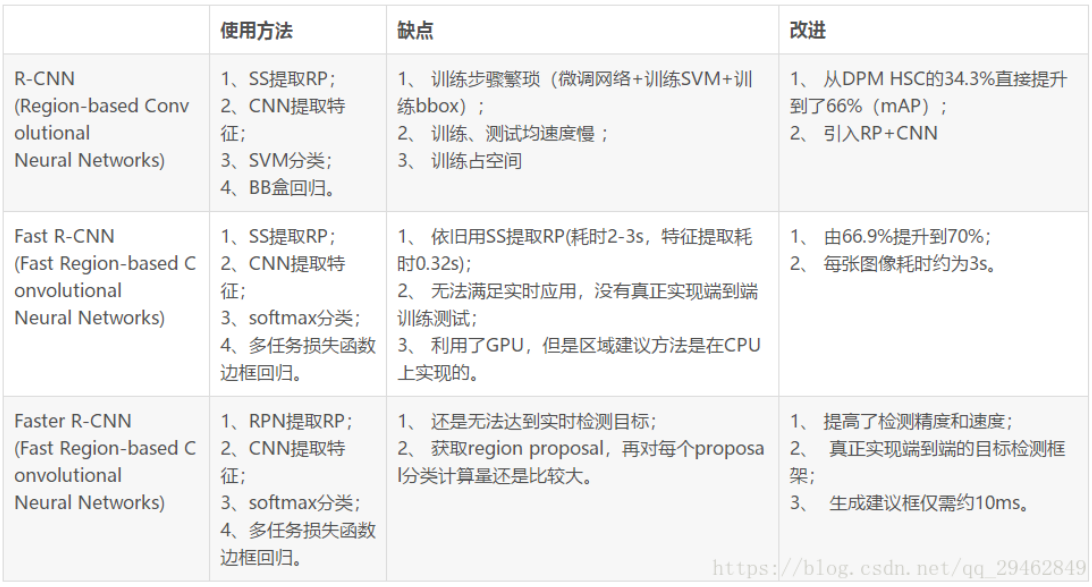
**候选区域方法效果对比图**

基于这种初步处理方式，R-CNN 算法会生成 2000 个候选区域，每个区域会被固定成 227*227,227*227 大小送入 CNN 分类器中，提取出 4096 个特征，这些特征被送入一个多类别的 SVM 分类器来预测该区域中含有每种物体的概率。此时，定位的准确性就显得尤为重要，所以 R-CNN 算法引入了一个边框回归模型，用以提高定位精确度。

显然，这种方法将 2000 个候选区域依次输入 CNN 分类器来提取特征，依然很麻烦。Fast R-CNN 解决了这个问题。Fast R-CNN 直接将图片传入一个多层的

全连接网络（为了保证精度，在此处不由于卷积神经网络而减少参数传入）提取整个图像的特征，然后使用划分候选区域的方法对得到的卷积特征图像（Conv feature map）进行划分。划分出的区域与其对应的特征图会被裁剪为特征图块用于目标检测任务中。这样的流程使得 Fast-RCNN 不会重复提取特征，以显著减少处理时间。

以上两种 RCNN 系列算法都依赖于外部的候选区域方法（如：选择性搜索），这限制了 RCNN 系列算法的运算速度。以 Fast-RCNN 为例，算法预测需要约 2.3 秒，而这其中生成 2000 个 ROI（region of interest）需要约 2 秒。这说明外部候选区域方法已经大大限制了 RCNN 系列算法的运算速度，为了解决这一问题，Faster-RCNN 诞生了。Faster-RCNN 与 Fast-RCNN 结构大体相同，但 Faster 用内部深层网络代替了候选区域方法，候选区域网络（RPN）在生成 ROI 时效率更高，正是这一点，使得 Faster 的训练速度和运行速度都远超 **Fast**。



| | 使用方法 | 缺点 | 改进 |
|---|---|---|---|
| R-CNN (Region-based Convolutional Neural Networks) | 1、SS提取RP；<br>2、CNN提取特征；<br>3、SVM分类；<br>4、BB盒回归。 | 1、 训练步骤繁琐（微调网络+训练SVM+训练bbox）；<br>2、 训练、测试均速度慢；<br>3、 训练占空间 | 1、 从DPM HSC的34.3%直接提升到了66%（mAP）；<br>2、 引入RP+CNN |
| Fast R-CNN (Fast Region-based Convolutional Neural Networks) | 1、SS提取RP；<br>2、CNN提取特征；<br>3、softmax分类；<br>4、多任务损失函数边框回归。 | 1、 依旧用SS提取RP(耗时2-3s，特征提取耗时0.32s)；<br>2、 无法满足实时应用，没有真正实现端到端训练测试；<br>3、 利用了GPU，但是区域建议方法是在CPU上实现的。 | 1、 由66.9%提升到70%；<br>2、 每张图像耗时约为3s。 |
| Faster R-CNN (Fast Region-based Convolutional Neural Networks) | 1、RPN提取RP；<br>2、CNN提取特征；<br>3、softmax分类；<br>4、多任务损失函数边框回归。 | 1、 还是无法达到实时检测目标；<br>2、 获取region proposal，再对每个proposal分类计算量还是比较大。 | 1、 提高了检测精度和速度；<br>2、 真正实现端到端的目标检测框架；<br>3、 生成建议框仅需约10ms。 |

RCNN 系列算法的总结

## Part 2：测试表现

RCNN 系列的三种算法，在准确度方面的提升并不是很明显（因为它们的主体算法大致相同），但是他们的速度提升非常显著。Fast-RCNN 与 RCNN 相比，训练时间减少７．８倍，测试时间减少１４５倍。Faster 与 Fast 相比速度提高１０倍。但是事实上 RCNN 系列作为基于候选区域的目标检测器，其速度依旧远远小于单次目标检测器。

## Part３：代码解析

鉴于目前最好的 RCNN 系列算法是 Faster－RCNN，在这里截取 Faster 的代码加以解析：

数据载入部分：

```
import torch as t
from .voc_dataset import VOCBboxDataset
from skimage import transform as sktsf
from torchvision import transforms as tvtsf
from . import util
import numpy as np
from utils.config import opt


def inverse_normalize(img):
    if opt.caffe_pretrain:
        img = img + (np.array([122.7717, 115.9465,
102.9801]).reshape(3, 1, 1))
        return img[::-1, :, :]
    # approximate un-normalize for visualize
    return (img * 0.225 + 0.45).clip(min=0, max=1) * 255


def pytorch_normalze(img):
    """
    https://github.com/pytorch/vision/issues/223
    return appr -1~1 RGB
    """
    normalize = tvtsf.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
    img = normalize(t.from_numpy(img))
    return img.numpy()


def caffe_normalize(img):#During the training phase,img is flipped
to increase training data
    """
    return appr -125-125 BGR
    """
    img = img[[2, 1, 0], :, :]  # RGB-BGR
    img = img * 255
    mean = np.array([122.7717, 115.9465, 102.9801]).reshape(3, 1, 1)
    img = (img - mean).astype(np.float32, copy=True)
    return img


def preprocess(img, min_size=600, max_size=1000):#scaling
    """Preprocess an image for feature extraction.

    The length of the shorter edge is scaled
to :obj:`self.min_size`.
    After the scaling, if the length of the longer edge is longer
than
    :param min_size:
    :obj:`self.max_size`, the image is scaled to fit the longer edge
    to :obj:`self.max_size`.

    After resizing the image, the image is subtracted by a mean
image value
    :obj:`self.mean`.

    Args:
        img (~numpy.ndarray): An image. This is in CHW and RGB
format.
            The range of its value is :math:`[0, 255]`.
         (~numpy.ndarray): An image. This is in CHW and RGB format.
            The range of its value is :math:`[0, 255]`.

    Returns:
        ~numpy.ndarray:
        A preprocessed image.

    """
    C, H, W = img.shape
    scale1 = min_size / min(H, W)
    scale2 = max_size / max(H, W)
    scale = min(scale1, scale2)
    img = img / 255.
    img = sktsf.resize(img, (C, H * scale, W * scale), 
mode='reflect')
    # both the longer and shorter should be less than
    # max_size and min_size
    if opt.caffe_pretrain:
        normalize = caffe_normalize
    else:
        normalize = pytorch_normalze
    return normalize(img)


class Transform(object):

    def __init__(self, min_size=600, max_size=1000):
        self.min_size = min_size
        self.max_size = max_size

    def __call__(self, in_data):
        img, bbox, label = in_data
        _, H, W = img.shape
        img = preprocess(img, self.min_size, self.max_size)
        _, o_H, o_W = img.shape
        scale = o_H / H
        bbox = util.resize_bbox(bbox, (H, W), (o_H, o_W))

        # horizontally flip
        img, params = util.random_flip(
            img, x_random=True, return_param=True)
        bbox = util.flip_bbox(
            bbox, (o_H, o_W), x_flip=params['x_flip'])

        return img, bbox, label, scale


class Dataset:
    def __init__(self, opt):
        self.opt = opt
        self.db = VOCBboxDataset(opt.voc_data_dir)
        self.tsf = Transform(opt.min_size, opt.max_size)

    def __getitem__(self, idx):
        ori_img, bbox, label, difficult = self.db.get_example(idx)

        img, bbox, label, scale = self.tsf((ori_img, bbox, label))
        # TODO: check whose stride is negative to fix this instead
copy all
        # some of the strides of a given numpy array are negative.
        return img.copy(), bbox.copy(), label.copy(), scale

    def __len__(self):
        return len(self.db)


class TestDataset:
    def __init__(self, opt, split='test', use_difficult=True):
        self.opt = opt
        self.db = VOCBboxDataset(opt.voc_data_dir, split=split,
use_difficult=use_difficult)

    def __getitem__(self, idx):
        ori_img, bbox, label, difficult = self.db.get_example(idx)
        img = preprocess(ori_img)
        return img, ori_img.shape[1:], bbox, label, difficult

    def __len__(self):
        return len(self.db)
```

网络结构：

```
import numpy as np
from torch import nn
from model.vgg16 import decom_vgg16#CNN model of pre-training
from model.rpn import RegionProposalNetwork#RPN network
from model.roi_module import VGG16RoIHead#ROI and above netbook
from utils.config import opt
import torch as t
from utils import array_tool as at


import cupy as cp
from model.utils.nms import non_maximum_suppression
from model.utils.bbox_tools import loc2bbox

from torch.nn import functional as F

class FasterRCNN(nn.Module):
    def __init__(self, ratios=[0.5, 1, 2], anchor_scales=[0.5, 1, 
2], \
                loc_normalize_mean = (0., 0., 0., 0.), \
                loc_normalize_std = (0.1, 0.1, 0.2, 0.2)):
        super(FasterRCNN, self).__init__()

        # prepare
        extractor, classifier = decom_vgg16()
        """
        The model was divided into two parts: extractor and 
classifier
        The extractor parameter is fixed
        """
        rpn = RegionProposalNetwork(
            512, 512,
            ratios=ratios,
            scales=anchor_scales,
            feat_stride=16
        )

        head = VGG16RoIHead(
            n_class=20 + 1,
            roi_size=7,
            spatial_scale=(1. / 16),
            classifier=classifier.cuda()
        )
        self.extractor = extractor.cuda()
        self.rpn = rpn.cuda()
        self.head = head

        # mean and std
        self.loc_normalize_mean = loc_normalize_mean
        self.loc_normalize_std = loc_normalize_std


    @property
    def n_class(self):
        # Total number of classes including the background.
        return self.head.n_class

    def forward(self, x, scale=1.):

        img_size = x.shape[2:]
        h = self.extractor(x)
        rpn_locs, rpn_scores, rois, roi_indices, anchor = \
            self.rpn(h, img_size, scale)
        roi_cls_locs, roi_scores = self.head(
            h.cuda(), rois, np.array(roi_indices))
        return roi_cls_locs, roi_scores, rois, roi_indices


    def get_optimizer(self):
        """
        return optimizer, It could be overwriten if you want to
specify
        special optimizer
        """
        lr = opt.lr
        params = []
        for key, value in dict(self.named_parameters()).items():
            if value.requires_grad:
                if 'bias' in key:
                    params += [{'params': [value], 'lr': lr * 2,
'weight_decay': 0}]
                else:
                    params += [{'params': [value], 'lr': lr,
'weight_decay': opt.weight_decay}]
        # if opt.use_adam:
        #     self.optimizer = t.optim.Adam(params)
        # else:
        self.optimizer = t.optim.SGD(params, momentum=0.9)
        return self.optimizer


    def use_preset(self, preset):
        """Use the given preset during prediction.

        This method changes values of :obj:`self.nms_thresh` and
        :obj:`self.score_thresh`. These values are a threshold value
        used for non maximum suppression and a threshold value
        to discard low confidence proposals in :meth:`predict`,
        respectively.

        If the attributes need to be changed to something
        other than the values provided in the presets, please modify
        them by directly accessing the public attributes.
        Args:
            preset ({'visualize', 'evaluate'): A string to determine
the
                preset to use.

        """
        if preset == 'visualize':
            self.nms_thresh = 0.3
            self.score_thresh = 0.7
        elif preset == 'evaluate':
            self.nms_thresh = 0.3
            self.score_thresh = 0.05
        else:
            raise ValueError('preset must be visualize or evaluate')

    def _suppress(self, raw_cls_bbox, raw_prob):
        bbox = list()
        label = list()
        score = list()
        # skip cls_id = 0 because it is the background class
        for l in range(1, self.n_class):
            cls_bbox_l = raw_cls_bbox.reshape((-1, self.n_class,
4))[:, l, :]
            prob_l = raw_prob[:, l]
            mask = prob_l > self.score_thresh
            cls_bbox_l = cls_bbox_l[mask]
            prob_l = prob_l[mask]
            keep = non_maximum_suppression(
                cp.array(cls_bbox_l), self.nms_thresh, prob_l)
            keep = cp.asnumpy(keep)
            bbox.append(cls_bbox_l[keep])
            # The labels are in [0, self.n_class - 2].
            label.append((l - 1) * np.ones((len(keep),)))
            score.append(prob_l[keep])
        bbox = np.concatenate(bbox, axis=0).astype(np.float32)
        label = np.concatenate(label, axis=0).astype(np.int32)
        score = np.concatenate(score, axis=0).astype(np.float32)
        return bbox, label, score

    def predict(self, imgs,sizes=None,visualize=False):
        """Detect objects from images.

        This method predicts objects for each image.
        Args:
            imgs (iterable of numpy.ndarray): Arrays holding images.
                All images are in CHW and RGB format
                and the range of their value is :math:`[0, 255]`.

        Returns:
           tuple of lists:
           This method returns a tuple of three lists,
           :obj:`(bboxes, labels, scores)`.

           * **bboxes**: A list of float arrays of shape :math:`(R,
4)`, \
               where :math:`R` is the number of bounding boxes in a
image. \
               Each bouding box is organized by \
               :math:`(y_{min}, x_{min}, y_{max}, x_{max})` \
               in the second axis.
           * **labels** : A list of integer arrays of
shape :math:`(R,)`. \
               Each value indicates the class of the bounding box. \
               Values are in range :math:`[0, L - 1]`,
where :math:`L` is the \
               number of the foreground classes.
           * **scores** : A list of float arrays of
shape :math:`(R,)`. \
               Each value indicates how confident the prediction is.

        """
        self.eval()
        self.use_preset('evaluate')
        if visualize:
            self.use_preset('visualize')
            prepared_imgs = list()
            sizes = list()
            for img in imgs:
                size = img.shape[1:]
                img = preprocess(at.tonumpy(img))
                prepared_imgs.append(img)
                sizes.append(size)
        else:
             prepared_imgs = imgs
        bboxes = list()
        labels = list()
        scores = list()
        for img, size in zip(prepared_imgs, sizes):
            img =
t.autograd.Variable(at.totensor(img).float()[None], volatile=True)
            scale = img.shape[3] / size[1]
            roi_cls_loc, roi_scores, rois, _ = self(img,
scale=scale)
            # We are assuming that batch size is 1.
            roi_score = roi_scores.data
            roi_cls_loc = roi_cls_loc.data
            roi = at.totensor(rois) / scale

            # Convert predictions to bounding boxes in image
coordinates.
            # Bounding boxes are scaled to the scale of the input
images.
            mean = t.Tensor(self.loc_normalize_mean).cuda(). \
                repeat(self.n_class)[None]
            std = t.Tensor(self.loc_normalize_std).cuda(). \
                repeat(self.n_class)[None]

            roi_cls_loc = (roi_cls_loc * std + mean)
            roi_cls_loc = roi_cls_loc.view(-1, self.n_class, 4)
            roi = roi.view(-1, 1, 4).expand_as(roi_cls_loc)
            cls_bbox = loc2bbox(at.tonumpy(roi).reshape((-1, 4)),
                                at.tonumpy(roi_cls_loc).reshape((-1,
4)))
            cls_bbox = at.totensor(cls_bbox)
            cls_bbox = cls_bbox.view(-1, self.n_class * 4)
            # clip bounding box
            cls_bbox[:, 0::2] = (cls_bbox[:, 0::2]).clamp(min=0,
max=size[0])
            cls_bbox[:, 1::2] = (cls_bbox[:, 1::2]).clamp(min=0,
max=size[1])

            prob = at.tonumpy(F.softmax(at.tovariable(roi_score),
dim=1))

            raw_cls_bbox = at.tonumpy(cls_bbox)
            raw_prob = at.tonumpy(prob)

            bbox, label, score = self._suppress(raw_cls_bbox,
raw_prob)
            bboxes.append(bbox)
            labels.append(label)
            scores.append(score)

        self.use_preset('evaluate')
        self.train()
        return bboxes, labels, scores
```
训练：
```
from collections import namedtuple

from torch.nn import functional as F
from model.utils.roi_sample import ProposalTargetCreator
from model.utils.rpn_gt_loc_label import AnchorTargetCreator

from torch import nn
import torch as t
from torch.autograd import Variable
from utils import array_tool as at

from utils.config import opt

LossTuple = namedtuple('LossTuple',
                       ['rpn_loc_loss',
                        'rpn_cls_loss',
                        'roi_loc_loss',
                        'roi_cls_loss',
                        'total_loss'
                        ])


class FasterRCNNTrainer(nn.Module):
    """wrapper for conveniently training. return losses

    The losses include:

    * :obj:`rpn_loc_loss`: The localization loss for \
        Region Proposal Network (RPN).
    * :obj:`rpn_cls_loss`: The classification loss for RPN.
    * :obj:`roi_loc_loss`: The localization loss for the head
module.
    * :obj:`roi_cls_loss`: The classification loss for the head
module.
    * :obj:`total_loss`: The sum of 4 loss above.

    Args:
        faster_rcnn (model.FasterRCNN):
            A Faster R-CNN model that is going to be trained.
    """

    def __init__(self, faster_rcnn):
        super(FasterRCNNTrainer, self).__init__()

        self.faster_rcnn = faster_rcnn
        self.rpn_sigma = opt.rpn_sigma
        self.roi_sigma = opt.roi_sigma

        # target creator create gt_bbox gt_label etc as training
targets.
        self.anchor_target_creator = AnchorTargetCreator()
        self.proposal_target_creator = ProposalTargetCreator()

        self.loc_normalize_mean = faster_rcnn.loc_normalize_mean
        self.loc_normalize_std = faster_rcnn.loc_normalize_std

        self.optimizer = self.faster_rcnn.get_optimizer()

    def forward(self, imgs, bboxes, labels, scale):
        """Forward Faster R-CNN and calculate losses.

        Here are notations used.

        * :math:`N` is the batch size.
        * :math:`R` is the number of bounding boxes per image.

        Currently, only :math:`N=1` is supported.

        Args:
            imgs (~torch.autograd.Variable): A variable with a batch
of images.
            bboxes (~torch.autograd.Variable): A batch of bounding
boxes.
                Its shape is :math:`(N, R, 4)`.
            labels (~torch.autograd..Variable): A batch of labels.
                Its shape is :math:`(N, R)`. The background is
excluded from
                the definition, which means that the range of the
value
                is :math:`[0, L - 1]`. :math:`L` is the number of
foreground
                classes.
            scale (float): Amount of scaling applied to
                the raw image during preprocessing.

        Returns:
            namedtuple of 5 losses
        """
        n = bboxes.shape[0]
        if n != 1:
            raise ValueError('Currently only batch size 1 is 
supported.')

        _, _, H, W = imgs.shape
        img_size = (H, W)

        features = self.faster_rcnn.extractor(imgs)
        """
        Input img to extractor to get features
        """
        rpn_locs, rpn_scores, rois, roi_indices, anchor = \
            self.faster_rcnn.rpn(features, img_size, scale)

        # Since batch size is one, convert variables to singular 
form
        bbox = bboxes[0]
        label = labels[0]
        rpn_score = rpn_scores[0]
        rpn_loc = rpn_locs[0]
        roi = rois

        # Sample RoIs and forward
        # it's fine to break the computation graph of rois, 
        # consider them as constant input
        sample_roi, gt_roi_loc, gt_roi_label = 
self.proposal_target_creator(
            roi,
            at.tonumpy(bbox),
            at.tonumpy(label),
            self.loc_normalize_mean,
            self.loc_normalize_std)
        """
```

Enter rois、bbox、label to proposal_target_creator to get sample_roi、gt_roi_loc、gt_roi_label.

The implication of this step is to get an ROI that is proportionately positive or negative.

```
        """
        # NOTE it's all zero because now it only support for batch=1
now
        sample_roi_index = t.zeros(len(sample_roi))
        roi_cls_loc, roi_score = self.faster_rcnn.head(
            features,
            sample_roi,
            sample_roi_index)

        # ------------------ RPN losses -------------------#
        gt_rpn_loc, gt_rpn_label = self.anchor_target_creator(
            at.tonumpy(bbox),
            anchor,
            img_size)
        gt_rpn_label = at.tovariable(gt_rpn_label).long()
        gt_rpn_loc = at.tovariable(gt_rpn_loc)
        rpn_loc_loss = _fast_rcnn_loc_loss(
            rpn_loc,
            gt_rpn_loc,
            gt_rpn_label.data,
            self.rpn_sigma)
        """
        Given rpn_loc , you need to get the real gt_rpn_loc and
gt_rpn_label from anchor and bbox.
        The loss calculation at this location only considers the
foreground , so it is sufficient to calculate L1-LOSS according to
rpn_loc、gt_rpn_loc               and gt_rpn_label.
        """
        # NOTE: default value of ignore_index is -100 ...
        rpn_cls_loss = F.cross_entropy(rpn_score,
gt_rpn_label.cuda(), ignore_index=-1)
        """
        Calculate the cross entropy of dichotomies according to
rpn_score and gt_rpn_label.
        """
        # ------------------ ROI losses (fast rcnn loss) -----------
--------#
        n_sample = roi_cls_loc.shape[0]
        roi_cls_loc = roi_cls_loc.view(n_sample, -1, 4)
        roi_loc = roi_cls_loc[t.arange(0, n_sample).long().cuda(), \
                              at.totensor(gt_roi_label).long()]
        gt_roi_label = at.tovariable(gt_roi_label).long()
        gt_roi_loc = at.tovariable(gt_roi_loc)
        roi_loc_loss = _fast_rcnn_loc_loss(
            roi_loc.contiguous(),
            gt_roi_loc.float(),
            gt_roi_label.data,
            self.roi_sigma)
        """
        Known roi_loc,we have got gt_roi_loc and gt_roi_label when
we go through sample roi.
        We can compute L1-LOSS based on roi_loc,gt_roi_loc and
gt_roi_label.
        """
        roi_cls_loss = nn.CrossEntropyLoss()(roi_score,
gt_roi_label.cuda())
        """
        Calculate the cross entropy of multiple classifications
according to roi_score and gt_roi_label.
        """
        losses = [rpn_loc_loss, rpn_cls_loss, roi_loc_loss,
roi_cls_loss]
        losses = losses + [sum(losses)]

        return LossTuple(*losses)

    def train_step(self, imgs, bboxes, labels, scale):
        self.optimizer.zero_grad()
        losses = self.forward(imgs, bboxes, labels, scale)
        losses.total_loss.backward()
        self.optimizer.step()
        return losses

def _smooth_l1_loss(x, t, in_weight, sigma):
    sigma2 = sigma ** 2
    # print ("------------")
    # print ("in_weight: ", in_weight)
    # print ("------------")
    # print ("x: ", x)
    # print ("------------")
    # print ("t: ", t)
    # print ("------------")
    t = t.float()
    diff = in_weight * (x - t)
    abs_diff = diff.abs()
    flag = (abs_diff.data < (1. / sigma2)).float()
    flag = Variable(flag)
    y = (flag * (sigma2 / 2.) * (diff ** 2) +
         (1 - flag) * (abs_diff - 0.5 / sigma2))
    return y.sum()


def _fast_rcnn_loc_loss(pred_loc, gt_loc, gt_label, sigma):
    in_weight = t.zeros(gt_loc.shape).cuda()
    # Localization loss is calculated only for positive rois.
    # NOTE:  unlike origin implementation,
    # we don't need inside_weight and outside_weight, they can
calculate by gt_label
    in_weight[(gt_label > 0).view(-1,
1).expand_as(in_weight).cuda()] = 1
    loc_loss = _smooth_l1_loss(pred_loc, gt_loc,
Variable(in_weight), sigma)
    # Normalize by total number of negtive and positive rois.
    loc_loss /= (gt_label >= 0).sum()  # ignore gt_label==-1 for
rpn_loss
    return loc_loss
```

测试:

```
    import numpy as np
    from torch import nn
    from model.vgg16 import decom_vgg16
    from model.rpn import RegionProposalNetwork
    from model.roi_module import VGG16RoIHead
    from utils.config import opt
    import torch as t
    from utils import array_tool as at



    import cupy as cp
    from model.utils.nms import non_maximum_suppression
    from model.utils.bbox_tools import loc2bbox

    from torch.nn import functional as F

    class FasterRCNN(nn.Module):
        def __init__(self, ratios=[0.5, 1, 2], anchor_scales=[0.5,
1, 2], \
                        loc_normalize_mean = (0., 0., 0., 0.), \
                        loc_normalize_std = (0.1, 0.1, 0.2, 0.2)):
            super(FasterRCNN, self).__init__()
            # prepare
            extractor, classifier = decom_vgg16()
            rpn = RegionProposalNetwork(
                512, 512,
                ratios=ratios,
                scales=anchor_scales,
                feat_stride=16
            )

            head = VGG16RoIHead(
                n_class=20 + 1,
                roi_size=7,
                spatial_scale=(1. / 16),
                classifier=classifier.cuda()
            )
            self.extractor = extractor.cuda()
            self.rpn = rpn.cuda()
            self.head = head

            # mean and std
            self.loc_normalize_mean = loc_normalize_mean
            self.loc_normalize_std = loc_normalize_std
        @property
        def n_class(self):
            # Total number of classes including the background.
            return self.head.n_class

        def forward(self, x, scale=1.):

            img_size = x.shape[2:]

            h = self.extractor(x)
            rpn_locs, rpn_scores, rois, roi_indices, anchor = \
                self.rpn(h, img_size, scale)
            roi_cls_locs, roi_scores = self.head(
                h.cuda(), rois, np.array(roi_indices))
            return roi_cls_locs, roi_scores, rois, roi_indices


        def get_optimizer(self):
            """
            return optimizer, It could be overwriten if you want to
specify
            special optimizer
```

```
            """
            lr = opt.lr
            params = []
            for key, value in dict(self.named_parameters()).items():
                if value.requires_grad:
                    if 'bias' in key:
                        params += [{'params': [value], 'lr': lr * 2,
'weight_decay': 0}]
                    else:
                        params += [{'params': [value], 'lr': lr,
'weight_decay': opt.weight_decay}]
            # if opt.use_adam:
            #     self.optimizer = t.optim.Adam(params)
            # else:
            self.optimizer = t.optim.SGD(params, momentum=0.9)
            return self.optimizer


        def use_preset(self, preset):
            """Use the given preset during prediction.

            This method changes values of :obj:`self.nms_thresh` and
            :obj:`self.score_thresh`. These values are a threshold
value
            used for non maximum suppression and a threshold value
            to discard low confidence proposals in :meth:`predict`,
            respectively.

            If the attributes need to be changed to something
            other than the values provided in the presets, please
modify
            them by directly accessing the public attributes.

            Args:
                preset ({'visualize', 'evaluate'): A string to
determine the
                    preset to use.

            """
            if preset == 'visualize':
                self.nms_thresh = 0.3
                self.score_thresh = 0.7
            elif preset == 'evaluate':
                self.nms_thresh = 0.3
                self.score_thresh = 0.05
            else:
                raise ValueError('preset must be visualize or
evaluate')

        def _suppress(self, raw_cls_bbox, raw_prob):
            bbox = list()
            label = list()
            score = list()
            # skip cls_id = 0 because it is the background class
            for l in range(1, self.n_class):
                cls_bbox_l = raw_cls_bbox.reshape((-1, self.n_class,
4))[:, l, :]
                prob_l = raw_prob[:, l]
                mask = prob_l > self.score_thresh
                cls_bbox_l = cls_bbox_l[mask]
                prob_l = prob_l[mask]
                keep = non_maximum_suppression(
                    cp.array(cls_bbox_l), self.nms_thresh, prob_l)
                keep = cp.asnumpy(keep)
                bbox.append(cls_bbox_l[keep])
                # The labels are in [0, self.n_class - 2].
                label.append((l - 1) * np.ones((len(keep),)))
                score.append(prob_l[keep])
            bbox = np.concatenate(bbox, axis=0).astype(np.float32)
            label = np.concatenate(label, axis=0).astype(np.int32)
            score = np.concatenate(score, axis=0).astype(np.float32)
            return bbox, label, score

        def predict(self, imgs,sizes=None,visualize=False):
            """Detect objects from images.

            This method predicts objects for each image.

            Args:
                imgs (iterable of numpy.ndarray): Arrays holding
images.
                    All images are in CHW and RGB format
                    and the range of their value is :math:`[0,
255]`.

            Returns:
               tuple of lists:
               This method returns a tuple of three lists,
               :obj:`(bboxes, labels, scores)`.

               * **bboxes**: A list of float arrays of
shape :math:`(R, 4)`, \
                   where :math:`R` is the number of bounding boxes
in a image. \
                   Each bouding box is organized by \
                   :math:`(y_{min}, x_{min}, y_{max}, x_{max})` \
                   in the second axis.
               * **labels** : A list of integer arrays of
shape :math:`(R,)`. \
                   Each value indicates the class of the bounding
box. \
                   Values are in range :math:`[0, L - 1]`,
where :math:`L` is the \
                   number of the foreground classes.
               * **scores** : A list of float arrays of
shape :math:`(R,)`. \
                   Each value indicates how confident the prediction
is.

            """
            self.eval()
            self.use_preset('evaluate')
            if visualize:
                self.use_preset('visualize')
                prepared_imgs = list()
                sizes = list()
                for img in imgs:
                    size = img.shape[1:]
                    img = preprocess(at.tonumpy(img))
                    prepared_imgs.append(img)
                    sizes.append(size)
            else:
                 prepared_imgs = imgs
            #Image preprocessing
            bboxes = list()
            labels = list()
            scores = list()
            for img, size in zip(prepared_imgs, sizes):
                img =
t.autograd.Variable(at.totensor(img).float()[None], volatile=True)
                scale = img.shape[3] / size[1]
                roi_cls_loc, roi_scores, rois, _ = self(img,
scale=scale)
                # We are assuming that batch size is 1.
                roi_score = roi_scores.data
                roi_cls_loc = roi_cls_loc.data
                roi = at.totensor(rois) / scale

                # Convert predictions to bounding boxes in image
coordinates.
                # Bounding boxes are scaled to the scale of the
input images.
                mean = t.Tensor(self.loc_normalize_mean).cuda(). \
                    repeat(self.n_class)[None]
                std = t.Tensor(self.loc_normalize_std).cuda(). \
                    repeat(self.n_class)[None]

                roi_cls_loc = (roi_cls_loc * std + mean)
                roi_cls_loc = roi_cls_loc.view(-1, self.n_class, 4)
                #Position correction ang score of ROI obtained by
head network.
                #input feature 、rois and roi_indices to get
roi_cls_loc and roi_score.
                roi = roi.view(-1, 1, 4).expand_as(roi_cls_loc)
                cls_bbox = loc2bbox(at.tonumpy(roi).reshape((-1,
4)),

at.tonumpy(roi_cls_loc).reshape((-1, 4)))
                cls_bbox = at.totensor(cls_bbox)
                cls_bbox = cls_bbox.view(-1, self.n_class * 4)
                #input roi_cls_loc、roi_score、rois,bbox was
predicted by nms and other mathods.
                # clip bounding box
                cls_bbox[:, 0::2] = (cls_bbox[:, 0::2]).clamp(min=0,
max=size[0])
                cls_bbox[:, 1::2] = (cls_bbox[:, 1::2]).clamp(min=0,
max=size[1])

                prob =
at.tonumpy(F.softmax(at.tovariable(roi_score), dim=1))

                raw_cls_bbox = at.tonumpy(cls_bbox)
                raw_prob = at.tonumpy(prob)
                bbox, label, score = self._suppress(raw_cls_bbox,
raw_prob)
                bboxes.append(bbox)
                labels.append(label)
                scores.append(score)

            self.use_preset('evaluate')
            self.train()
            return bboxes, labels, scores
```

## Part 4：运行结果